Programming Languages

Modules and Exceptions

CSCI-GA.2110-001
Summer 2012
Programs are built out of components called modules.

Each module:

- has a public interface that defines entities exported by the module
- may include other (private) entities that are not exported
- may depend on the entities defined in the interface of another module (weak external coupling)
- should define a set of logically related entities (strong internal coupling)
What is a module?

- different languages use different terms
- different languages have different semantics for this construct (sometimes very different)
- a module is somewhat like a record, but with an important distinction:
  - **record** $\implies$ consists of a set of names called *fields*, which refer to values in the record.
  - **module** $\implies$ consists of a set of names, which can refer to values, types, routines, other language-specific entities, and possibly other modules
Language constructs for modularity

Issues:

- public interface
- private implementation
- dependencies between modules
- naming conventions of imported entities
- relationship between modules and files
- access control: module controls whether a client can access its contents
- closed module: names must be explicitly imported from outside the module
- open module: outside names are accessible inside module (no explicit import)
Language choices

- **Ada**: package declaration and body, `with` and `use` clauses, renamings
- **C**: header files, `#include` directives
- **C++**: header files, `#include` directives, namespaces, `using` declarations/directives, namespace alias definitions
- **Java**: packages, `import` statements
- **ML**: signature, `structure` and `functor` definitions
package Queues is
    Size: constant Integer := 1000;

    type Queue is private; -- information hiding

    procedure Enqueue (Q: in out Queue, Elem: Integer);
    procedure Dequeue (Q: in out Queue; Elem: out Integer);
    function Empty (Q: Queue) return Boolean;
    function Full (Q: Queue) return Boolean;
    function Slack (Q: Queue) return Integer;
    -- overloaded operator "=":
    function "=" (Q1, Q2: Queue) return Boolean;

private
    ... -- concern of implementation, not of package client
end Queues;
package Queues is
    ... -- visible declarations
private
    type Storage is
        array (Integer range <>) of Integer;
    type Queue is record
        Front: Integer := 0; -- next elem to remove
        Back: Integer := 0; -- next available slot
        Contents: Storage (0 .. Size-1); -- actual contents
        Num: Integer := 0;
    end record;
end Queues;
package body Queues is
  procedure Enqueue (Q: in out Queue;
      Elem: Integer) is
  begin
    if Full(Q) then
      -- need to signal error: raise exception
    else
      Q.Contents(Q.Back) := Elem;
    end if;
    Q.Num := Q.Num + 1;
    Q.Back := (Q.Back + 1) mod Size;
  end Enqueue;
function Empty (Q: Queue) return Boolean is begin
    return Q.Num = 0;  -- client cannot access
    -- Num directly
end Empty;

function Full (Q: Queue) return Boolean is begin
    return Q.Num = Size;
end Full;

function Slack (Q: Queue) return Integer is begin
    return Size - Q.Num;
end Slack;
function "=" (Q1, Q2 : Queue) return Boolean is
begin
   if Q1.Num /= Q2.Num then
      return False;
   else
      for J in 1 .. Q1.Num loop
         -- check corresponding elements
         if Q1.Contents((Q1.Front + J - 1) mod Size) /=
            Q2.Contents((Q2.Front + J - 1) mod Size)
         then
            return False;
         end if;
      end loop;
      return True; -- all elements are equal
   end if;
end "="; -- operator "/=" implicitly defined
-- as negation of "="
with Queues; use Queues; with Text_IO;

procedure Test is
  Q1, Q2: Queue; -- local objects of a private type
  Val : Integer;
begin
  Enqueue(Q1, 200); -- visible operation
  for J in 1 .. 25 loop
    Enqueue(Q1, J);
    Enqueue(Q2, J);
  end loop;
  Dequeue(Q1, Val); -- visible operation
  if Q1 /= Q2 then
    Text_IO.Put_Line("lousy implementation");
  end if;
end Test;
package body holds bodies of subprograms that implement interface
package may not require a body:

package Days is
  type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

  subtype Weekday is Day range Mon .. Fri;

  Tomorrow: constant array (Day) of Day := (Tue, Wed, Thu, Fri, Sat, Sun, Mon);

  Next_Work_Day: constant array (Weekday) of Weekday := (Tue, Wed, Thu, Fri, Mon);
end Days;
Visible entities can be denoted with an expanded name:

```haskell
with Text_IO;
...
Text_IO.Put_Line("hello");
```

use clause makes name of entity directly usable:

```haskell
with Text_IO; use Text_IO;
...
Put_Line("hello");
```

renames clause makes name of entity more manageable:

```haskell
with Text_IO;
package T renames Text_IO;
...
T.Put_Line("hello");
```
with Queues;

procedure Test is
  Q1, Q2: Queues.Queue;
begin
  if Q1 = Q2 then ...
    -- error: "=" is not directly visible
    -- must write instead: Queues."="(Q1, Q2)

Two solutions:

- import all entities:
  use Queues;

- import operators only:
  use type Queues.Queue;
late addition to the language
an entity requires one or more declarations and a single definition
a namespace declaration can contain both, but definitions may also be given separately

```cpp
// in .h file
namespace util {
    int f (int); /* declaration of f */
}

// in .cpp file
namespace util {
    int f (int i) {
        // definition provides body of function
        ...
    }
}
```
Files have semantic significance: `#include` directives mean textual substitution of one file in another. The convention is to use header files for shared interfaces.

```cpp
#include <iostream> // import declarations

int main () {
    std::cout << "C++ is really different" << std::endl;
    return 0;
}
```
namespace stack { // in file stack.h
    void push (char);
    char pop ();
}

#include "stack.h" // import into client file

void f () {
    stack::push ('c');
    if (stack::pop () != 'c') error("impossible");
}
#include "stack.h" // import declarations

namespace stack { // the definition
    const unsigned int MaxSize = 200;
    char v[MaxSize];
    unsigned int numElems = 0;

    void push (char c) {
        if (numElems >= MaxSize)
            throw std::out_of_range("stack\noverflow");
        v[numElems++] = c;
    }

    char pop () {
        if (numElems == 0)
            throw std::out_of_range("stack\nunderflow");
        return v[--numElems];
    }
}
namespace queue { // works on single queue
    void enqueue (int);
    int dequeue ();
}

#include "queue.h" // in client file
using queue::dequeue; // selective: a single entity
void f () {
    queue::enqueue(10); // prefix needed for enqueue
    queue::enqueue(-999);
    if (dequeue() != 10) // but not for dequeue
        error("buggy implementation");
}
Wholesale import: the using directive

```c
#include "queue.h" // in client file

using namespace queue; // import everything

void f () {
    enqueue(10); // prefix not needed
    enqueue(-999);
    if (dequeue() != 10) // for anything
        error("buggy implementation");
}
```
Sometimes, we want to qualify names, but with a shorter name.

In Ada:

```ada
package PN renames A.Very_Long.Package_Name;
```

In C++:

```cpp
namespace pn = a::very_long::package_name;
```

We can now use `PN` as the qualifier instead of the long name.
When an unqualified name is used as the postfix-expression in a function call (\texttt{expr.call}), other namespaces not considered during the usual unqualified lookup (\texttt{basic.lookup.unqual}) may be searched; this search depends on the types of the arguments.

For each argument type $T$ in the function call, there is a set of zero or more associated namespaces to be considered. The set of namespaces is determined entirely by the types of the function arguments. \texttt{typedef} names used to specify the types do not contribute to this set.

The set of namespaces are determined in the following way:
If T is a primitive type, its associated set of namespaces is empty.
If T is a class type, its associated namespaces are the namespaces in which the class and its direct and indirect base classes are defined.
If T is a union or enumeration type, its associated namespace is the namespace in which it is defined.
If T is a pointer to U, a reference to U, or an array of U, its associated namespaces are the namespaces associated with U.
If T is a pointer to function type, its associated namespaces are the namespaces associated with the function parameter types and the namespaces associated with the return type. [recursive]
namespace NS
{
    class A {};
    void f( A *&, int ) {};
}

int main()
{
    NS::A *a;
    f( a, 0 );    // calls NS::f
}
an external declaration for a variable indicates that the entity is defined elsewhere

```c
extern int x; // will be found later
```

a function declaration indicates that the body is defined elsewhere

```c
extern int x; // in some other file
```

an entity can only be *defined* once

missing/multiple definitions cannot be detected by the compiler: link-time errors
package structure parallels file system
- a package corresponds to a directory
- a class is compiled into a separate object file
- each class declares the package in which it appears (open structure)

```java
package polynomials;
class poly {
    // in file .../alg/polynomials/poly.java
}

package polynomials;
class iterator {
    // in file .../alg/polynomials/iterator.java
}
```

Default: anonymous package in current directory.
dependencies indicated with `import` statements:

```java
import java.awt.Rectangle; // declared in java.awt

import java.awt.*; // import all classes // in package
```

- no syntactic sugar across packages: use expanded names
- none needed in same package: all classes in package are directly visible to each other
There are three entities:

- **signature**: an interface
- **structure**: an implementation
- **functor**: a parameterized structure

A **structure** implements a **signature** if it defines everything mentioned in the **signature** (in the correct way).
An ML *signature* specifies an interface for a module.

```ml
signature STACKS =
sig
  type stack
  exception Underflow
  val empty : stack
  val push : char * stack -> stack
  val pop : stack -> char * stack
  val isEmpty : stack -> bool
end
```
A *structure* provides an implementation.

```ml
structure Stacks : STACKS =
struct
    type stack = char list
    exception Underflow
    val empty = [ ]
    val push = op::
    fun pop (c::cs) = (c, cs)
        | pop [] = raise Underflow
    fun isEmpty [ ] = true
        | isEmpty _ = false
end
```
A **functor** creates a structure from a structure.

```ml
signature TOTALORDER = sig
  type element;
  val lt : element * element -> bool;
end;

functor MakeBST(Lt: TOTALORDER): sig
  type 'label btree;
  exception EmptyTree;
  val create : Lt.element btree;
  val lookup : Lt.element * Lt.element btree -> bool;
  val insert : Lt.element * Lt.element btree -> Lt.element btree;
```
val deletemin : Lt.element btree \rightarrow
Lt.element * Lt.element btree;
val delete : Lt.element * Lt.element btree
\rightarrow Lt.element btree;
end = struct
open Lt;
datatype 'label btree = Empty |
    Node of 'label * 'label btree * 'label btree;
val create = Empty;
fun lookup(x, Empty) = ...;
fun insert(x, Empty) = ...;
exception EmptyTree;
fun deletemin(Empty) = ...;
fun delete(x,Empty) = ...;
end;
structure String : TOTALORDER =
  struct
    type element = string;
    fun lt(x,y) =
      let
        fun lower(nil) = nil |
          lower(c::cs) =
            (Char.toLower c)::lower(cs);
        in
          implode(lower(explode(x))) <
          implode(lower(explode(y)))
        end;
    end;
  end;

structure StringBST = MakeBST(String);
Comparisons

<table>
<thead>
<tr>
<th></th>
<th>Ada</th>
<th>C++</th>
<th>Java</th>
<th>ML</th>
</tr>
</thead>
<tbody>
<tr>
<td>used to avoid name clashes</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>access control</td>
<td>✗</td>
<td>weak</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>is closed</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

Relation between interface and implementation:

- **Ada**: one package (interface) ⇔ one package body

- **ML**:
  - one signature *can be implemented by* many structures
  - one structure *can implement* many signatures
Exceptions

General mechanism for handling abnormal conditions

One way to improve robustness of programs is to handle errors. How can we do this?

We can check the result of each operation that can go wrong (e.g., popping from a stack, writing to a file, allocating memory).

Unfortunately, this has a couple of serious disadvantages:

1. it is easy to forget to check
2. writing all the checks clutters up the code and obfuscates the common case (the one where no errors occur)

Exceptions let us write clearer code and make it easier to catch errors.
Predefined exceptions in Ada

- **Defined in Standard:**
  - `Constraint_Error`: value out of range
  - `Program_Error`: illegality not detectable at compile-time: unelaborated package, exception during finalization, etc.
  - `Storage_Error`: allocation cannot be satisfied (heap or stack)
  - `Tasking_Error`: communication failure

- **Defined in Ada.IO_Exceptions:**
  - `Data_Error`, `End_Error`, `Name_Error`, `Use_Error`, `Mode_Error`, `Status_Error`, `Device_Error`
Any begin-end block can have an exception handler:

```plaintext
procedure Test is
  X: Integer := 25;
  Y: Integer := 0;
begin
  X := X / Y;
exception
  when Constraint_Error =>
    Put_Line("did you divide by 0?");
  when others =>
    Put_Line("out of the blue!");
end;
```
function Get_Data return Integer is
  X: Integer;
begin
  loop
    begin
      Get(X);
      return X;  -- if got here, input is valid,
                  -- so leave loop
    exception
      when others =>
        Put_Line("input must be integer, try again");
        -- will restart loop to wait for a good input
    end;
  end loop;
end;
package Stacks is
  Stack_Empty: exception;
...
end Stacks;

package body Stacks is
  procedure Pop (X: out Integer;
                 From: in out Stack) is
  begin
    if Empty(From)
      then raise Stack_Empty;
    else ...
  end Pop;
...
end Stacks;
The scope of exceptions

- an exception has the same visibility as other declared entities: to handle an exception it must be visible in the handler (e.g., caller must be able to see `Stack_Empty`).
- an `others` clause can handle unnamed exceptions

```pl
when others =>
  Put_Line("disaster\somewhere");
raise;  -- propagate exception,
        -- program will terminate
```
How to propagate an exception:

1. When an exception is raised, the current sequence of statements is abandoned (e.g., current `Get` and `return` in example)
2. Starting at the current frame, if we have an exception handler, it is executed, and the current frame is completed.
3. Otherwise, the frame is discarded, and the enclosing dynamic scopes are examined to find a frame that contains a handler for the current exception (want dynamic as opposed to static scopes because those are values that caused the problem).
4. If no handler is found, the program terminates.

Note: The current frame is never resumed.
an Ada exception is a label, not a value: we cannot declare exception variables and then assign to them

but an exception *occurrence* is a value that can be stored and examined

an exception occurrence may include additional information: source location of occurrence, contents of stack, etc.

predefined package *Ada.Exceptions* contains needed machinery
package Ada.Exceptions is
    type Exception_Id is private;
    type Exception_Occurrence is limited private;

    function Exception_Identity (X: Exception_Occurrence)
        return Exception_Id;
    function Exception_Name (X: Exception_Occurrence)
        return String;

    procedure Save_Occurrence
        (Target: out Exception_Occurrence;
         Source: Exception_Occurrence);
    procedure Raise_Exception (E: Exception_Id;
                              Message: in String := "")

    ...
end Ada.Exceptions;
begin
  ...
exception
  when Expected: Constraint_Error =>
    -- Expected has details
    Save_Occurrence(Event_Log, Expected);

  when Trouble: others =>
    Put_Line("unexpected \" &
              Exception_Name(Trouble) &
              "\" raised\";)
    Put_Line("shutting \"down\";
    raise;
end;
Exceptions in C++

- similar runtime model,...
- but exceptions are bona-fide values,
- handlers appear in `try/catch` blocks

```cpp
try {
    some_complex_calculation();
} catch (const RangeError& e) {
    // RangeError might be raised
    // in some_complex_calculation
    cerr << "oops\n";
} catch (const ZeroDivide& e) {
    // same for ZeroDivide
    cerr << "why is denominator zero?\n";
}
```
The program throws an object. There is nothing needed in the declaration of the type to indicate it will be used as an exception.

```c
struct ZeroDivide {
    int lineno;
    ZeroDivide (...) { ... } // constructor
    ...
};
...
if (x == 0)
    throw ZeroDivide(...); // call constructor
    // and go
A handler names a class, and can handle an object of a derived class as well:

```cpp
class Matherr { }; // a bare object, no info
class Overflow : public Matherr {...};
class Underflow : public Matherr {...};
class ZeroDivide : public Matherr {...};

try {
    weatherPredictionModel(...);
} catch (const Overflow& e) {
    // e.g., change parameters in caller
} catch (const Matherr& e) {
    // Underflow, ZeroDivide handled here
} catch (...) {
    // handle anything else (ellipsis)
}
```
Exceptions in Java

■ Model and terminology similar to C++:
  ◆ exceptions are objects that are thrown and caught
  ◆ try blocks have handlers, which are examined in succession
  ◆ a handler for an exception can handle any object of a derived class

■ Differences:
  ◆ all exceptions are extensions of predefined class Throwable
  ◆ checked exceptions are part of method declaration
  ◆ the finally clause specifies clean-up actions
  ■ in C++, cleanup actions are idiomatically done in destructors
System errors are extensions of Error and RuntimeException; these are unchecked exceptions. Examples: ClassCastException, NullPointerException, OutOfMemoryError.

All other exception classes are checked. These exceptions must be either handled or declared in the method that throws them; this is checked by the compiler.
If a method might throw an exception, callers should know about it.

```
public void replace (String name,
                      Object newValue) throws NoSuch
{
    Attribute attr = find(name);
    if (attr == null) throw new NoSuch(name);
    newValue.update(attr);
}
```
Some cleanups must be performed whether the method terminates normally or throws an exception.

```java
public void parse (String file) throws IOException {
    BufferedReader input =
        new BufferedReader(new FileReader(file));
    try {
        while (true) {
            String s = input.readLine();
            if (s == null) break;
            parseLine(s);  // may fail somewhere
        }
    } finally {
        if (input != null) input.close();
    }  // regardless of how we exit
}
Exceptions in ML

- runtime model similar to Ada/C++/Java
- **exception** is a single type (like a **datatype** but dynamically extensible)
- declaring new sorts of exceptions:
  ```plaintext
  exception StackUnderflow
  exception ParseError of { line: int, col: int }
  ```
- raising an exception:
  ```plaintext
  raise StackUnderflow
  raise (ParseError { line = 5, col = 12 })
  ```
- handling an exception:
  ```plaintext
  expr₁ handle pattern => expr₂
  ```
  If an exception is raised during evaluation of `expr₁`, and `pattern` matches that exception, `expr₂` is evaluated instead
exception DivideByZero

fun f i j = 
    if j <> 0
    then i div j
    else raise DivideByZero

(f 6 2
 handle DivideByZero => 42)  (* evaluates to 3 *)

(f 4 0
 handle DivideByZero => 42)  (* evaluates to 42 *)

Typing issues:
- the type of the body and the handler must be the same
- the type of a \texttt{raise} expression can be \textit{any type}
  (whatever type is appropriate is chosen)